RUST

Patrick Gilmore
Department of Computer Science
University of Arizona
Tucson, Arizona 85721
pmgilmore41@email.arizna.edu

Jack Lyons
Department of Computer Science
University of Arizona
Tucson, Arizona 85721
jacklyons@email.arizona.edu

April 1, 2020

ABSTRACT

1 Introduction

2 History

2.1 Why?

Most systems applications are written in C or C++ but as many masters of C know, C can have tons of Pitfalls. To solve this problem Rust was developed so that developers could write high Performance code that would normally need to be written in C or C++. Now with Rust developers need to worry much less about memory failure and code failing as a result of a segmentation fault.

2.2 Who?

The development of Rust was started by Graydon Hoare, a self proclaimed language engineer, in 2006 has a personal project. Down the line Mozilla took an interest in the project and put together a team to assist in the development of Rust. Mozilla continued to back Rust with the aim to rebuild their browser stack. To do so Mozilla aimed to replace the difficult C++ code with something that was easier to debug, write, and maintain efficiency.

2.3 Rust Today

Rust is still going strong to this day as indicated by stackoverflow.com, a popular site for developers to learn and ask questions about technology. In fact stackoverflow has voted Rust to by the most loved language 4 years in a row. Rust's drive to reduce the number of pains that developers face while writing system code in C++ while minimizing the consequences has been pulling in frustrated developers for years. So long as a developer does not need to make an application that goes way deep into the fundamentals of a system, Rust will permit easy convenient code.

3 Control Structures

Given that most large applications will require some degree of control flow, it would only make sense that rust would include some of its own. Since rust is derived from languages like C and C++ some of the structures may look familiar. That is not to say that rust has no unique features of its own!

3.1 If-else

Of all the control structures found across the many programming languages in existence, the if-else structure is the most common. For those less familiar, the if-else structure is used to evaluate the truth (aka Boolean value) of some statement and determine was block of code to execute on. This can help programs only perform certain tasks when the correct conditions are met and avoid code that could crash the application if the conditions are not met (like a null). Rust takes a pretty standard approach to how these structures look.

```
let x = 5

if x == 5 {
    println!("x is 5");
}
else {
    println!("x is not 5");
}
```

The structure takes some statement, determines the truth of it and jumps to the block of code that is appropriate. In the case above the statement is true as x is equal to 5 so it will jump to the block of code starting just below the if clause. It is important to note that in rust the statement MUST be a Boolean or at least evaluate to a Boolean. Programming languages like C allow the clause to be a number but if that was attempted in rust, the program would fail.
Rust also supports the common else if statement which allows a program to step through a number of possible options while only choosing one.

```
let x = ??

if x == 5 {
    println!("x is 5");
}
else if x < 5 {
    println!("x is less than 5");
}
else {
    println!("x greater than 5");
}
```

The above code will start by checking if x is 5 and if it is not then it will check if x is less than five and finally assume it must be greater than 5.
A very interesting feature of the Rust if-else structure is its ability to assign variables on the clause. We likely see statements like the following in the more common coding languages:

```
let mut x = 5

if x == 5 {
    x = 10
}
else {
    x = 2
}
```

But in rust we have the ability to assign x from the if statement itself!

```
let x = 5

x = if x == 5{
    10
}
else {
    2
}
```

The only catch to this neat little trick is that in each block, the value assigned to x needs to be of compatible types. So, if we had tried to assign x to "two" in the else block an error would have triggered.

3.2 Loop

Rust comes with an interesting control structure called loop. Loop will repeat until the program is stopped manually or until a condition within the loop block is met. This is not like for loops you would see in other languages where the loop is executed however many times and the condition is looping on changes is constant manner at the end of each iteration (i++). Rust does include a while loop which achieves its purpose in a very similar manner so whats the point. Well Rust allows you to set a return value to the loop.

```
let mut count = 0;
let returnCode = loop {
    count = count + 1;

    if count = 5 {
        break count;
    }
}
```

The above code would perform the loop until the count variable reaches 5 and then returns that number into the new variable returnCode. This may prove useful to those who need a variable quickly figured out based on dynamic data but then never change afterwards.

3.3 while

This may come as a very refreshing structure to those learning rust as it is pretty much the same as you would find in any other language. The while loop combines the power of loop, if, else and break and proves to be a useful tool in most programs. At the beginning of the structure, the programmer sets a condition for the loop to end on and follows that with the body of code they want repeated. It can allow loops to be shortened much more if the return of the statements in the code block are unneeded.

```
let mut count = 0;
while count != 5 {
    count = count + 1;
    println!("Count is {}", count);
}
```

In the above code we don't care about what count is, we just want the program to print what it is at each step so that mean the return code is useless to us. A perfect time to us the while loop!! It is important to remember to have a section within the block that changes the state of the loop so that at some point the program to exit and not get caught in an infinite loop.

3.4 for

Programmers will often find themselves needing to iterate over a data set and determine some meaningful result from the data within. This is where the for structure comes in! Much like loop and while, for performs a block of code on repeat until a condition is met forcing it to exit. Unlike the other two, for must be over an iterable type like a list. It is still possible to perform loops similar to what we saw in the while section using for. It simply requires the use of ranges in Rust. Since the range is iterable, Rust can step through it until it reaches then end.

```
let mut lst = [1,2,3,4,5];
for elem in lst.iter() {
    println!("Count is {}", elem);
}

for num in (1..5) {
    println!("Count is {}", num);
}
```

The loops above both do the same thing but the first is over a list which could contain in theory anything while the second uses a range between 1 and 5. Since for tends to achieve a large subsection of what programmers need to get done, it is the most popular of the looping structures.

3.5 match

Another of the major control structures available in Rust is match. Match is very similar to the switch case structure you would find in C or Java. The idea behind it is that you provide a case (something to match to) and a series of rules for the possible cases you care about. A use for this would be to have a list of rules based on student grades. The case would be a letter grade and the definition for the case could be the GPA equivalence. This is very convenient for what would normally be massive if else blocks as they do the same thing but with much less syntax required. As we mentioned this is very similar to the switch case structures found elsewhere but with major differences in the syntax.

```
let x = 2;
match x {
    1 => {println!("1");},
    2 => {println!("2");},
    _ => {println!("unknown");}
}
```

The above is equivalent to:

```
switch(x) {
    case 1:
        print("1");
        break;
    case 2:
        print("2");
        break;
    default:
        print("unknown");
}
```

Each case is the value before the '=>' and the definition for the case follows. It should be noted that the brackets in the definition section are not required if the definition is only one statement. To represent the default case, the case you go to if all else do not match is represented by and underscore which means anything. The Rust syntax also does not require the break to signal the end of a case rather it uses a comma.

4 Data Types

5 Subprograms

6 Summary

[?] [?] [?] [?] [?] [?] [?]